14

Systems
Optimization
Laboratory

LEVEL II

Department of Operations Research
Stanford University
Stanford, CA 94305

14

SYSTEMS OPTIMIZATION LABORATORY
DEPARTMENT OF OPERATIONS RESEARCH
Stanford University
Stanford, California
94305

SOLVING STAIRCASE LINEAR PROGRAMS BY THE SIMPLEX METHOD, 2: PRICING

by

Robert Fourer

TECHNICAL REPORT SOL 79-19
November 1979

Research and reproduction of this report were partially supported by the Department of Energy Contract DE-AS03-76-SF00326 PA# DE-AT-03-76ER72018; the Office of Naval Research Contract N00014-75-C-0267; the National Science Foundation Grants MCS76-20019 A01 and ENG77-06761.

INTRODUCTION

The goals of this paper are those of its predecessor [2]: to solve staircase-structured linear programs faster through adaptation of the algorithms of the modern simplex method. The means are quite different, however. Whereas [2] concentrated on "inversion" algorithms that factorize the basis and solve linear systems, the present paper looks at "pricing" algorithms that find a variable to enter the basis at each iteration.

Pricing involves two sorts of activities: computation of reduced costs that determine which variables are eligible to enter the basis, and selection of an entering variable from among those eligible. Thus an implementation of pricing in the simplex method requires both computation and selection algorithms. Either sort of algorithm may be adapted to staircase structure.

This paper begins with a short general review of staircase linear programs. Sections 2 and 3 then look at the computation algorithms, after which Sections 4 and 5 deal with selection algorithms. Staircase adaptations of both kinds of algorithms are proposed and evaluated.

Section 6 reports extensive computational experience with the preceding proposals. The staircase computation algorithms for pricing are found to offer modest but consistent savings. Staircase selection algorithms, properly chosen, produce more spectacular results: the number of iterations, time per iteration, or both may be reduced substantially by comparison with standard methods.

Overall the prospects for staircase adaptation of the simplex method appear highly promising. When the methods of this paper are combined with those of [2] in the most efficient way, one may expect savings of 50% or more for many different kinds of staircase linear programs.

1. STAIRCASE LINEAR PROGRAMS

This section summarizes the treatment of staircase structures developed in [1,2], with special attention to properties important in pricing.

Formulation

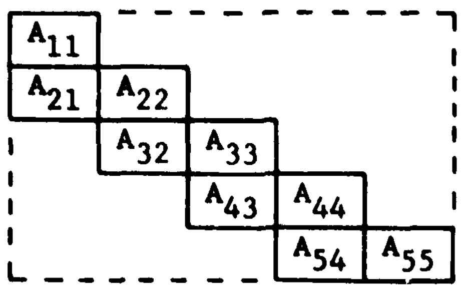
Staircase linear programs (LPs), as defined in [2], share two simple characteristics: their variables fall into some sequence of disjoint groups; and their constraints relate only variables within adjacent groups. Usually the sequence of groups corresponds to a sequence of times, so that variables in a group are said to represent activities of one _period_. Constraints thus indicate how activities in one period are related to activities in the next.

A constraint is said to be in period ℓ if it contains variables of period ℓ but not of later periods. Typically some constraints involve only variables of period ℓ, while others relate variables of periods ℓ and $\ell-1$; the latter are _linking_ constraints, whereas the former are _non-linking_. Analogously, linking variables appear in constraints of periods ℓ and $\ell+1$, while non-linking variables appear only in constraints of period ℓ.

A more general approach defines a staircase LP to be _of order_ r if its constraints relate variables that are at most r periods apart. Many of the ideas of this paper are applicable to staircase LPs of any order. However, the emphasis is on first-order staircases as defined above: these have the simplest and strongest structure, and so are best suited to special techniques.

Staircase matrices

Following [2], the matrix of constraint coefficients of a staircase linear program is a staircase matrix. Its nonzero elements are confined to certain submatrices centered roughly on and just off the diagonal--as, for example,



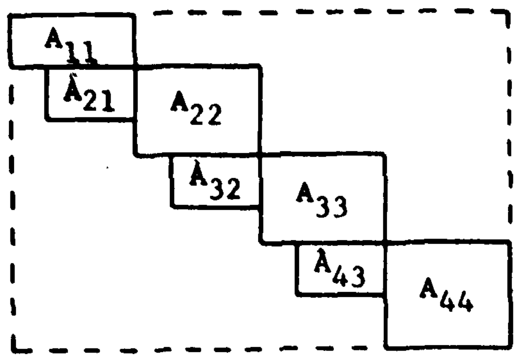
Formally, one partitions the rows of an $m \times n$ matrix A into t disjoint subsets, and the columns into t disjoint subsets, so that A is partitioned into t^2 submatrices, or "blocks":

$$A_{ij} \qquad i = 1,\ldots,t;\ j = 1,\ldots,t$$

A is lower staircase (as above) if $A_{ij} = 0$ except for $i = j$ and $i = j+1$. A is upper staircase if $A_{ij} = 0$ except for $i = j$ and $i = j-1$.

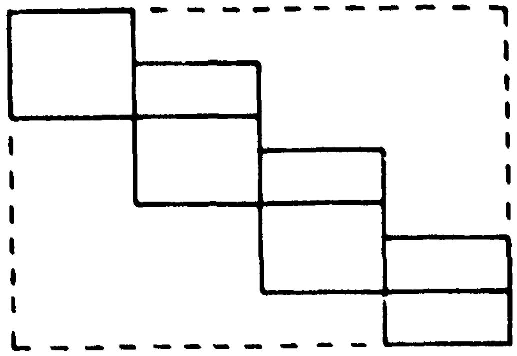
By analogy with staircase LPs, rows in the ith partition of a staircase matrix A are called period-i rows, and columns in the jth partition are called period-j columns. If a period-i row has nonzero elements in blocks $A_{i,i-1}$ and A_{ii}, it is a linking row; if it has nonzeroes only in A_{ii} it is a non-linking row. Similarly, a period-j column that has nonzeroes in A_{jj} and $A_{j+1,j}$ is a linking column, whereas one that has nonzeroes in A_{jj} only is a non-linking column.

It may be assumed, without loss of generality, that all staircase LPs have a constraint matrix A in standard form: A is lower staircase, and the diagonal blocks $A_{\ell\ell}$ have no all-zero rows or columns. If A is permuted to put the linking rows of each period first, and the linking columns of each period last, then it also has the following reduced form:



The intersection of period-k linking rows and period-(k-1) linking columns is the reduced block $\hat{A}_{k,k-1}$.

If the linking rows of every period i are switched to period i-1 then A gains an alternative row-upper-staircase form:



Switching the linking columns of period j to period $j+1$ gives a different, column-upper-staircase form. Thus a staircase A in reduced standard form embodies three staircases--lower, row-upper, and column-upper--corresponding to three different choices of where the periods begin and end.

Staircase bases

Any basis B of a staircase linear program necessarily inherits a staircase structure from the constraint matrix A: B's staircase blocks, $B_{t,t-1}$ and B_{tt}, may be taken as the sub-blocks of $A_{t,t-1}$ and A_{tt} that lie in the basic columns. If A has a reduced form, $\dot{B}_{t,t-1}$ may likewise be taken as the basic part of $\dot{A}_{t,t-1}$.

Either B_{tt} or $\dot{B}_{t,t-1}$ may be zero along some linking row i if it happens that, in A_{tt} or $\dot{A}_{t,t-1}$, all the nonzeroes along row i are on non-basic columns. Thus the inherited staircase of B need not be in standard or reduced form, even if A is.

Henceforth any $m \times m$ basis B will be assumed to have the staircase form inherited from A. The number of rows of B in period i will be denoted m_i, and the number of columns of B in period j will be denoted n_j; the respective numbers of linking rows and columns will be $\hat{m}_i$ and $\hat{n}_j$. For the row-upper-staircase form, the number of rows in period i will be m^i, and for the column-upper-staircase form the number of columns in period j will be n^j. Necessarily $\sum m_i = \sum m^i = \sum n_j = \sum n^j = m$, and $\hat{m}_i \leq m_i$, $\hat{n}_j \leq n_j$.

Balance constraints and square sub-staircases

Since the basis B is nonsingular, it must obey the "balance constraints" developed in [1]. In summary, these restrict the excess of rows over columns in each period, individually and cumulatively, as follows:

$$0 \leq \sum_1^{\ell} (n_i - m_i) \leq \min(\hat{m}_{\ell+1}, \hat{n}_{\ell}) \quad , \qquad \ell = 1,\dots, t-1$$

$$-\min(\hat{m}_k, \hat{n}_{k-1}) \leq \sum_k^{\ell} (n_i - m_i) \leq \min(\hat{m}_{\ell+1}, \hat{n}_{\ell}) \quad , \qquad k,\ell = 1,\dots, t-1$$

$$-\min(\hat{m}_k, \hat{n}_{k-1}) \leq \sum_k^{t} (n_i - m_i) \leq 0 \ , \qquad k = 2,\dots, t$$

In words, the cumulative imbalance between rows and columns in periods k through ℓ is bounded by the smaller dimension of $\hat{B}_{k,k-1}$ and the smaller dimension of $\hat{B}_{\ell+1,\ell}$. Hence these constraints are quite strict when there are relatively few linking rows or columns.

The first constraint above may also be written as the following three inequalities:

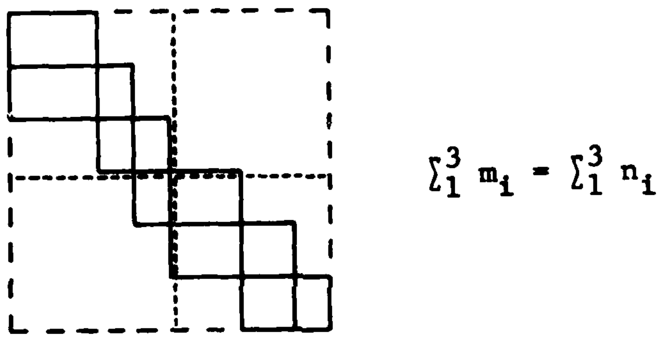
$$\sum_1^{\ell} n_i \geq \sum_1^{\ell} m_i$$

$$\sum_1^{\ell} n_i \leq \sum_1^{\ell} m^i$$

$$\sum_1^{\ell} n^i \leq \sum_1^{\ell} m_i$$

These say that the first ℓ periods of the lower staircase cannot have more rows than columns, while the first ℓ periods of the associated row-upper or column-upper staircase cannot have more columns than rows.

All three of these relations are equalities when $\ell = t$, since B is square. It can also happen that equality is achieved for some $\ell < t$. For example, if $\sum_1^\ell m_i = \sum_1^\ell n_i$, B must look something like this:



The rows and columns of periods 1 through ℓ form a <u>square sub-staircase</u>, as do the rows and columns of periods $\ell+1$ through t; they are linked only by nonzero elements in the off-diagonal block $\hat{B}_{\ell+1,\ell}$. In a similar way an equality $\sum_1^\ell n_i = \sum_1^\ell m^i$ implies a pair of square sub-staircases within the row-upper staircase form, and $\sum_1^\ell n^i = \sum_1^\ell m_i$ implies the same for the column-upper form.

Generally B may exhibit any or all of these three kinds of equalities, and each may hold for several values of $\ell < t$. If p different such equalities hold, then B breaks into $p+1$ disjoint square sub-staircases. Square sub-staircases can have a strong effect upon the iteration path of the simplex method, as Section 4 will show.

2. PRICING IN THE SIMPLEX METHOD

Each iteration of the simplex method begins with the choice of a nonbasic variable to enter the basis. The operations involved in making this choice are collectively referred to as pricing. Timings of staircase LPs in [2] show that pricing is invariably an expensive part of the simplex method, typically one-third to two-thirds of the total cost.

This section surveys current techniques of pricing, to set the stage for discussion of staircase pricing in the sequel.

To keep things simple it will be assumed that the problem is to maximize $c^T x$, that x is subject only to explicit constraints $Ax = b$ and implicit nonnegativity $(x \geq 0)$, and that a feasible basis is at hand. Straightforward extensions permit implementation of the composite simplex method [16,20] for infeasible bases, and of implicit upper and lower bounds of all sorts.

The constraint matrix A will be taken to include the objective as row 0; a_j will be the jth column of A. B will represent a basis, and $\tilde{B}$ the succeeding basis; a tilde will also indicate other quantities that have changed with the basis. A unit column with a 1 in row j will be written e_j.

Choosing a variable to enter the basis

Central to all pricing techniques are the reduced costs, d_j, $j = 1, \ldots, n$. If the jth variable is nonbasic, $d_j > 0$ implies that the objective will increase when j is brought into the basis (at a positive value), while $d_j < 0$ implies that the objective will decrease when j

enters the basis. Thus the simplex method can be guaranteed to find a maximum provided that, at each iteration, a variable j having $d_j > 0$ is chosen to enter the basis.

At most iterations there are many "eligible" nonbasic variables whose reduced costs are positive. Any pricing technique, therefore, must incorporate some criterion for choosing among the eligible nonbasics.

Early experiments [12,21] showed that the number of iterations in the simplex method is highly sensitive to the pricing criterion. Although it suffices to choose an eligible nonbasic at random, considerably fewer iterations are required when the chosen variable q has a maximum reduced cost:

$$d_q = \max_j d_j \qquad (1)$$

This criterion produces the greatest improvement in the objective per unit change in the activity of the nonbasic variable. Not surprisingly, the number of iterations is further reduced when q is chosen to maximize total improvement in the objective at the current iteration.

Still better criteria have been developed by extending (1) in a different sense. Criterion (1) can be looked at as follows [5]: it chooses to move along an edge of the feasible region whose gradient is steepest in the subspace of the nonbasic variables. A superior criterion chooses instead a steepest edge in the space of all variables. Writing $\bar{a}_{ij}$ ($= [B^{-1}a_j]_i$) for the change in the ith basic variable per unit change in nonbasic variable j, such a "steepest-edge" criterion chooses q so that

$$d_q/\sqrt{1+\sum \bar{a}_{iq}^2} = \max_j \left[d_j/\sqrt{1+\sum \bar{a}_{ij}^2} \right] \qquad (2)$$

In computational tests [5,6,12,21] this criterion and its variants have consistently yielded the fewest iterations.

Computational considerations

The efficacy of a pricing criterion must be balanced against the computational effort required. Generally a "better" criterion cuts the number of iterations but requires more computation per iteration, so that it may or may not be less expensive overall. For example, to implement the greatest-total-improvement criterion one must first determine

$$\Delta_j = \min_i [x_i/\bar{a}_{ij}]$$

for every nonbasic j such that $d_j > 0$; then the incoming variable q must satisfy

$$\Delta_q d_q = \max_j \Delta_j d_j$$

These tests require a prohibitive amount of computation for all but the smallest LPs.

Of all criteria that employ reduced costs, the gre test-reduced-cost criterion (1) is certainly the simplest to implement: it requires only the d_j's. Moreover, d_j can be computed efficiently in any of three ways:

(a) Solve $B^T\pi = e_0$ for a vector of <u>prices</u>, π. Then compute $d_j = \pi^T a_j$.

(b) Update π from the previous iteration: if nonbasic variable q replaced the pth basic variable, with pivot element α, first solve $B^T v = e_p$ for v, then compute $\tilde{\pi} = \pi - (d_q/\alpha)v$. Find $d_j = \tilde{\pi}^T a_j$ as before.

(c) Update d_j from the previous iteration: solve for v as in (b), then compute $\tilde{d}_j = d_j - (d_q/\alpha)v^T a_j$.

On average (c) requires the least computational effort and (a) the most, as explained in detail in [4,18]. However, (a) may afford considerable savings if not all d_j's are computed at each iteration (as in "partial pricing" discussed below); in addition, with (a) not all of π need be computed for staircase problems (as explained in Section 3). Method (b) also allows partial pricing but must compute all of π; partial pricing is impractical with (c), which must find every d_j at each iteration. For staircase problems a hybrid of (a) and (b) may be desirable, as Section 3 shows.

Steepest-edge criteria (2) replace d_j by a more complicated function,

$$d_j/\sqrt{1 + \sum \bar{a}_{ij}^2}$$

Two practical implementations compute or approximate this function as follows:

(d) Goldfarb [5] stores an additional set of weights, $\gamma_j = 1 + \sum \bar{a}_{ij}^2$, which are updated at each iteration. The incoming column may then be chosen to maximize the square of the above function, d_j^2/γ_j, over all nonbasic j such that $d_j > 0$.

(e) Harris [6] takes a similar approach, but employs weights $T_j \approx \sqrt{1 + \sum \bar{a}_{ij}^2}$ that are updated at most iterations and are reset to 1 when they become too inaccurate. The incoming variable is chosen to maximize d_j/T_j.

The advantage of (e) is that it requires only slightly more computation than (a)-(c), whereas (d) must solve an extra linear system with B^T and must compute extra inner products. On the other hand, (d)'s more accurate criterion tends to find an optimum in fewer iterations. Both (d) and (e) must update all weights at each iteration by solving $B^T v = e_p$, computing $v^T a_j$ for nonbasic j, and additional steps; the d_j's are thus cheaply updated at the same time by method (c). However, any efficiencies of method (a) or (b) through partial pricing are lost.

<u>Partial pricing</u>

All of the above methods involve an inner product to compute each d_j: either $\pi^T a_j$ for (a)-(b), or $v^T a_j$ for (c)-(e). Vector a_j is usually very sparse, so that only its nonzero elements and their row indices need be stored. Hence any one of these inner products can be computed cheaply. Nevertheless, the total cost of these products for all d_j's may be substantial, especially if the LP has considerably more variables than constraints.

Partial pricing attempts to speed up method (a) or (b) by considering only some of the nonbasic variables as candidates to enter the basis at each iteration. Only d_j's for these candidate nonbasics are needed, and hence fewer inner products are computed. If the set of candidates is kept small, the cost of an iteration can be markedly reduced. However, the number of iterations tends to be greater under partial pricing since superior potential candidates are often left out of the candidate set.

Partial pricing is thus essentially a matter of trading more iterations for less work per iteration. A good partial-pricing scheme chooses candidate sets that make this tradeoff a favorable one. The simplest schemes partition the variables into a fixed collection of candidate sets that are priced in rotation; more sophisticated schemes determine the candidate sets dynamically, stopping when a "good enough" reduced cost has been found. Dynamic partial pricing may be viewed mathematically as a complicated optimal-stopping problem in which the distribution of the reduced costs is unknown and gradually changing. Some attempt has been made to solve this problem systematically [7]; for the most part, however, partial-pricing schemes have employed heuristics chosen because they seemed reasonable and worked adequately.

The number of implemented strategies for partial pricing is immense--probably no two large scale systems price in exactly the same way. If there are any common principles, they are to avoid stopping prematurely with too small a reduced cost, and to stop promptly once a big reduced cost is found. Measures of what is too small and what is big may be either absolute or relative: absolute tests compare reduced costs with threshold values that are set initially from experience or by a heuristic algorithm, and are updated as the magnitude of reduced costs declines; relative tests make comparisons only among reduced costs found at the current iteration.

Storage considerations and multiple pricing

Partial pricing is especially attractive when the columns a_j are not stored in high-speed memory, so that computing d_j involves reading a_j into memory as well as an inner product. To save reading costs, some implementations operate as follows: at "major" iterations a regular partial price is carried out, and a subset of attractive candidate columns is saved in high-speed memory; at subsequent "minor" iterations only this candidate subset is priced. This sort of "multiple" pricing may also be viewed as a partial-pricing scheme that gives preference to variables whose reduced costs were high in previous iterations.

Storage considerations have declined in importance with the advent of paged machines that can cheaply simulate large memory regions for LP codes. Interest in multiple pricing has thus declined as well, and the experiments in this paper use simpler partial-pricing schemes. Any of these schemes could be adapted, however, to admit multiple pricing.

It is possible that some versions of partial or multiple pricing are more economical for paged computers because they tend to access fewer different pages of memory per iteration. Economies of this sort would probably be small, however, and so for present purposes the effects of paging have been disregarded.

3. COMPUTING PRICES FOR STAIRCASE LPs

Consider now a linear program of staircase structure. Suppose that the reduced cost of nonbasic column j is to be computed by $d_j = \pi^T a_j$ in method (a) or (b) of the previous section. If a_j is from period ℓ then it has nonzero elements only on rows of periods ℓ and $\ell+1$; as a consequence, the inner product $\pi^T a_j$ requires only elements of π that correspond to period-ℓ and period-$(\ell+1)$ rows.

More generally, if a_j is a column from period ℓ or later, then its reduced cost can be calculated from only those elements of π that correspond to rows of period ℓ or later. This statement is true for higher-order staircases as well as first-order ones.

These facts would be of no practical importance if all d_j's were calculated at every iteration. However, under partial pricing most candidate sets will contain variables of only certain periods, and so at most iterations only part of π will really be needed. The cost of pricing might therefore be reduced if only the needed part of π were computed.

Selective computation of the price vector

In practice there is no efficient way to compute arbitrary elements of π independently of the other elements. Nevertheless, useful portions of π can be computed more cheaply than all of π, provided that the basis is arranged in an appropriate way.

Consider first method (a) of the preceding section, which finds π as the solution of $B^T \pi = e_0$. Current LP codes solve this system by a form of Gaussian elimination, as explained in detail in [2]. After a series of various computations (whose specifics are not important here) the elements of π are finally produced by a routine called BTRANL. Essentially BTRANL comprises a single main loop; each pass through this loop computes a new element of π from the previously-computed elements. Thus a portion of π can be computed by simply stopping BTRANL prematurely. If later more of π is needed, BTRANL can be restarted where it left off.

Unfortunately BTRANL cannot produce the elements of π in any desired order; instead it must compute π in a fixed order that corresponds to the ordering of B's rows for Gaussian elimination. Standard methods of elimination choose a row ordering for efficiency and numerical stability, without regard to the periods of the staircase. As a result BTRANL tends to produce elements of π from various periods indiscriminately, and it may be necessary to run most of BTRANL to compute all elements of π for even one period.

To compute portions of π usefully, therefore, B's ordering for Gaussian elimination must preserve the staircase structure. Two such orderings are described in [1,2]: both leave the rows of the basis in or nearly in period order. BTRANL then produces the elements of π in nearly reverse period order: $t, t-1, \ldots, 2, 1$.

With these staircase orderings it is practical to selectively compute π. If the partial-pricing scheme starts with columns from period ℓ, then BTRANL is called to compute π for periods $t, t-1, \ldots, \ell+1, \ell$ only. This portion of π will suffice so long as all candidates are in period ℓ or later. If some candidate falls in an earlier period, say k, then BTRANL is resumed where it left off to compute π for periods

$\ell-1, \ldots, k+1, k$. BTRANL may be restarted in this way several times if the candidate set includes successively earlier periods.

The savings in computing only part of π can be significant: BTRANL is inherently a relatively expensive routine and accounts for most of the cost of solving $B^T\pi = e_0$. A set of six test problems in [2] spent roughly 15-20% of their total time in BTRANL, or 20-25% of their time if only iterating routines were considered.

Actual savings necessarily depend on the chosen pricing scheme. If the candidate set usually contains variables from early periods then little will be gained; preferably the candidate set is confined to later periods at a good proportion of iterations. For best results one may use special staircase partial-pricing methods, which are the topic of Section 5.

Selective updating of the price vector

It can be faster to update π--by method (b) of Section 2--than to recompute π from $B^T\pi = e_0$. However, a full update requires a full solution to a system like $B^Tv = e_k$, and it will usually be cheaper to solve $B^T\pi = e_0$ for part of π than to solve $B^Tv = e_k$ for all of v. Thus full updating of π should be disadvantageous for staircase LPs under partial pricing.

Selective updating is a practical alternative. Suppose π is known for periods $\ell, \ldots, t$, while at the next iteration π will be needed for periods $k, \ldots, t$. If $k \geq \ell$, π can simply be updated: first $B^Tv = e_k$ is solved by BTRANL for petiods $t, \ldots, k$ only; then periods k through t of $\bar{\pi}$ are found by the updating formula

$$\grave{\pi}_i = \pi_i - (d_q/\alpha)v_i$$

On the other hand, if $k < \ell$ then π can be updated only as far back as period ℓ. The remainder of $\grave{\pi}$ must be found by solving $B^T\grave{\pi} = e_0$ in the usual way--except that BTRANL may <u>skip</u> the computation of $\grave{\pi}$ for periods $t,\ldots,\ell$. (Alternatively, if it is known that $k < \ell$ before pricing begins then it may be cheaper to drop the update step and just solve $B^T\grave{\pi} = e_0$ for periods $t,\ldots,k$ of $\grave{\pi}$.)

Selective updating is essentially a hybrid of methods (a) and (b) of Section 2. It might be possible to also include (c) in the hybrid, so that some d_j's are updated from v rather than being computed from π. However, it is not clear that the additional savings would be worth the extra complications.

In any event, the steepest-edge methods--(d) and (e)--must compute all of v to update their weights, regardless of which d_j's they examine. Hence these methods require a full BTRANL at every iteration, and cannot benefit from selective computation or updating of π.

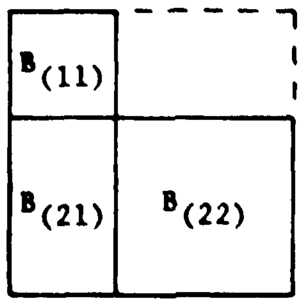
4. ITERATION PATHS OF STAIRCASE LPs

The choice of a variable to enter the basis also largely determines the variable that leaves. Thus different pricing techniques should be expected to produce different sorts of iteration paths in the simplex method.

For staircase LPs the connection between pricing and iteration path can be especially strong and clear. This section shows that, when the staircase basis has a certain sub-structure, an entering variable from a given period must determine a leaving variable from a certain range of periods. Moreover, the basic solution is unchanged outside of this range. These observations contribute to the design of staircase partial-pricing schemes in Section 5.

Restrictions on the outgoing column

Suppose first that the basis has a square lower-block-triangular form:



(3)

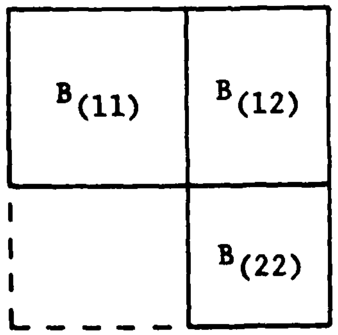
Assume further that the incoming column q is zero on all of the rows of $B_{(11)}$. Then it is easy to see that the outgoing column must be from $B_{(22)}$: otherwise $B_{(22)}$ would gain a column and the new basis would be singular. In short, if the incoming column belongs to the second block then so does the outgoing column.

Consider now the values of the basic variables. They satisfy $Bx = b$, where b is the LP's right-hand side; partitioning x and b in conformance with the blocks of B,

$$\begin{aligned} B_{(11)}x^{(1)} \qquad\qquad\qquad &= b^{(1)} \\ B_{(21)}x^{(1)} + B_{(22)}x^{(2)} &= b^{(2)} \end{aligned}$$

When column q above is brought into the basis, only $B_{(22)}$ is altered; consequently, basic variables $x^{(1)}$ are unchanged. In sum, if the incoming column belongs to the second block then the basic solution changes only in that block.

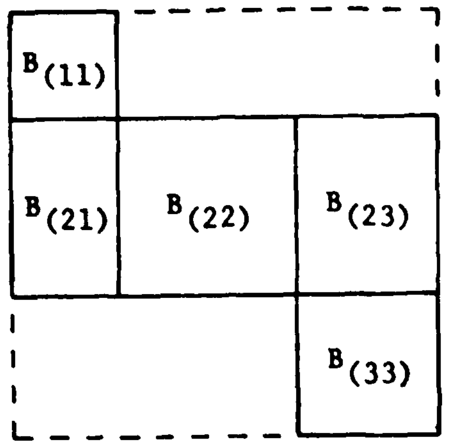
A parallel analysis applies to any square upper-block-triangular form of B:



(4)

If the incoming column belongs to the first block then so does the outgoing column, and the basic solution changes only in the first block.

Putting these observations together, B can be imagined to have a form like this:



(5)

where $B_{(\ell\ell)}$ are all square. $B_{(22)}$ is both part of a lower-triangular block (with $B_{(33)}$) and part of an upper-triangular block (with $B_{(11)}$). Thus if the incoming column belongs to the middle block $(B_{(22)})$ then so does the outgoing column, and the basic solution changes only in the middle block.

Restrictions on the outgoing column of a staircase basis

Typical bases have many block-triangular partitions, and this fact is used to advantage in sparse Gaussian elimination [1]. However, there is generally no clear relation between these blocks and the structure of the linear program. As a result there is no easy way to keep track of the blocks from iteration to iteration, and there is no reason to expect that an incoming column will belong to one block or another.

The situation is quite different when B has a staircase form. Then any square sub-staircase--as defined in Section 1--comprises a square diagonal block. Square sub-staircases are easily kept track of from iteration to iteration since they are defined by simple relations like $\sum_{i=1}^{\ell} m_i = \sum_{i=1}^{\ell} n_i$. Moreover, there is every reason to expect that an incoming variable may lie entirely with a square sub-staircase.

As a consequence stronger statements can be made about the iteration path of staircase LPs. Recall from Section 1 that lower square sub-staircases arise when $\sum_{i=1}^{\ell} m_i = \sum_{i=1}^{\ell} n_i$ for some $\ell < t$. In diagram (3) above, let $B_{(11)}$ represent the sub-staircase of the first ℓ periods, while $B_{(22)}$ is the sub-staircase of periods $\ell+1$ through t; $B_{(12)}$ contains the linking block $B_{\ell+1,\ell}$. Then any incoming column from periods $\ell+1$ through t must lie in $B_{(22)}$. The conclusions of the preceding subsection can now be applied to imply that

- If $\sum_{i=1}^{\ell} m_i = \sum_{i=1}^{\ell} n_i$, and if the incoming column is from periods $\ell+1,\ldots,t$, then the outgoing column is also from periods $\ell+1,\ldots,t$ and the activities of variables in periods $1,\ldots,\ell$ do not change.

Upper square sub-staircases arise similarly in connection with the row-upper and column-upper forms of B defined in Section 1. By analogous reasoning in conjunction with diagram (4), it can also be concluded that

- If $\sum_1^\ell m^i = \sum_1^\ell n_i$, and if the incoming column is from periods $1,\ldots,\ell$, then the outgoing column is also from periods $1,\ldots,\ell$ and the activities of variables in periods $\ell+1,\ldots,t$ do not change.

- If $\sum_1^\ell m_i = \sum_1^\ell n^i$, and if the incoming column is from upper-staircase periods $1,\ldots,\ell$, then the outgoing column is also from upper periods $1,\ldots,\ell$ and the activities of variables in upper periods $\ell+1,\ldots,t$ do not change.

(As defined in Section 1, the upper-staircase periods $1,\ldots,\ell$ comprise the corresponding lower-staircase periods less the period-ℓ linking columns.)

In the general case B may contain both upper and lower sub-staircases, producing a situation as in diagram (5). Roughly speaking, if periods $1,\ldots,k$ form a lower square sub-staircase and periods $\ell+1,\ldots,t$ form an upper one, then an incoming column from periods $k+1,\ldots,\ell$ yields an outgoing column from the same periods and leaves activities outside these periods unchanged.

Since staircase bases must be well-balanced (Section 1) the presence of square sub-staircases should not be unusual. Indeed, results of test runs suggest that bases exhibit several square sub-staircases more often than not. Furthermore, it must be kept in mind that the above statements apply to <u>any</u> staircase partitioning of the LP, not just to the staircase identified by the modeler. Hence a number of other square sub-staircases may go unidentified.

Prevalence of square sub-staircases is also suggested by the iteration paths of test runs: only infrequently is the outgoing column more than a few periods from the incoming one. For example, the six test problems of Section 6 (under full pricing) show the incoming and outgoing variables two or fewer periods apart in 59%, 74%, 85%, 94%, 97%, and 98% of a'l iterations.

Implications for pricing

Square sub-staircases may be viewed generally as creating barriers to pricing at certain periods. A lower sub-staircase in periods 1 through ℓ, for example, places a lower barrier at ℓ: if the incoming column is below the barrier then the outgoing column is also below the barrier, only basic activities below the barrier are changed, and the barrier remains at the next iteration. The basis and basic activities above the barrier can change only if the incoming column is above the barrier, and the barrier is removed only if additionally the outgoing column is below it.

These facts have important implications for partial pricing. So long as the candidate set lies in periods below a lower barrier, activities in periods above the barrier are unchanged. In effect, pricing below the barrier suboptimizes the LP in the below-barrier periods, while fixing the above-barrier periods. By contrast, pricing above the barrier optimizes all the periods and tends to break the barrier down.

Upper sub-staircases naturally have an analogous but opposite effect: they create upper barriers at certain periods. Pricing above an upper barrier suboptimizes the LP in above-barrier periods, while fixing the below-barrier periods.

In general a basis can have several upper and lower barriers. Thus pricing in any one period suboptimizes the periods between the nearest preceding lower barrier and the nearest succeeding upper barrier, fixing the others.

As long as the basis tends to have square sub-staircases, therefore, partial pricing will tend to promote suboptimization. This may be a good thing if the suboptima are near the true optimum, or a bad thing if the suboptima are far from optimal. Both extremes were observed in the computational experiments in Section 6.

Partial-pricing schemes can be devised either to encourage or discourage suboptimization. Pricing the same periods repeatedly tends to create barriers and suboptimize, while pricing throughout the matrix tends to do the opposite. It is also possible to keep track of the number of columns in each period so as to price where barriers will most likely be created or destroyed. Strategies along all of these lines are developed in the following section.

5. PARTIAL PRICING FOR STAIRCASE LPs

It is now clear--given the results of the two preceding sections --that any partial-pricing scheme for staircase LPs should distinguish among variables of different periods. Consequently all of the methods proposed below price essentially one period at a time. These methods differ considerably, however, in their choices of periods to price and in their stopping criteria.

Four general procedures for staircase pricing are presented first below. Subsequent subsections propose and evaluate specific variations on these procedures.

Simple pricing by period

All methods of partial pricing by period involve some ordering of the periods--$p_1^{(k)}$, $p_2^{(k)}, \ldots, p_t^{(k)}$--at each iteration k. Assume for the moment that such an ordering has been chosen, according to one of the principles suggested later in this section.

The most straightforward method first takes the nonbasics of period $p_1^{(k)}$ as the candidate set. If any of these variables has a favorable reduced cost, one having a largest reduced cost is chosen to enter the basis. Otherwise the nonbasics of period $p_2^{(k)}$ are added to the candidate set; if any of these has a favorable reduced cost, a best one is chosen. If necessary, the process repeats with $p_3^{(k)}$, $p_4^{(k)}, \ldots, p_t^{(k)}$, stopping when a favorable d_j is found in some period. If no favorable d_j is found, the current basis is optimal.

This procedure will be called <u>simple</u> pricing by period. A practical algorithm to carry it out is as follows:

<u>SIMPLE PRICING</u>:

```
1:  REPEAT FOR $\ell$ FROM 1 TO $t$:
        CHOOSE $q_\ell$ such that $d_{q_\ell} \geq d_j$ for all $j$ in period $p_\ell^{(k)}$
        IF $d_{q_\ell} > \tau^{(k)}$, select $q_\ell$ to enter basis; RETURN
2:  CHOOSE $q$ such that $d_q \geq d_{q_\ell}$ for $\ell = 1, \ldots, t$
    IF $d_q > \varepsilon$ : Select $q$ to enter basis; RETURN
3:  Declare basis optimal; RETURN
```

Two tolerances are employed. A fixed tolerance, ε, defines the smallest reduced cost that is considered different from zero. A dynamic tolerance, $\tau^{(k)} > \varepsilon$, defines an "acceptable" reduced cost: step 1 of the algorithm will not let variable q_ℓ enter the basis unless $d_{q_\ell} > \tau^{(k)}$. If $d_j \leq \tau^{(k)}$ for all j, then step 2 may select an entering column q having $\varepsilon < d_q \leq \tau^{(k)}$.

A subsidiary algorithm is required to update $\tau^{(k)}$ at each iteration. The general idea is to pick $\tau^{(k)}$ large enough that the chosen d_q is not too far from $\max_j d_j$, but small enough that only one or two periods are priced at most iterations. For all tests in Section 6, $\tau^{(k)}$ is updated in the following way:

```
    $\tau^{(1)} = 10^{30}$
1:  IF $d_q > \tau^{(k)}$:   $\tau^{(k+1)} = \frac{1}{2}(\tau^{(k)} + \gamma d_q)$
2:  IF $d_q \leq \tau^{(k)}$:   $\tau^{(k+1)} = \gamma d_q$
```

In effect $\tau^{(k)}$ is a fraction γ of a running average of the chosen reduced costs d_q. To get a good starting value, $\tau^{(1)}$ is set to "infinity," forcing $\tau^{(2)} = \gamma d_q$ in step 2; subsequently step 1 updates the moving average, and step 2 is invoked only if all reduced costs fall below $\tau^{(k)}$. For a range of test problems (Section 6) a γ of 0.2 or 0.5 usually gave best results.

<u>Simple pricing with threshold</u>

Simple pricing by period might be improved by adding a more sophisticated stopping rule. For example, a "desirable" reduced cost can be defined by a value $T^{(k)} \geq \tau^{(k)}$ at each iteration: any variable with $d_j \geq T^{(k)}$ is immediately brought into the basis without further pricing. If all reduced costs in a period are less than $T^{(k)}$, then a variable with $d_j > \tau^{(k)}$ may be chosen as before.

This strategy will be called simple pricing with <u>threshold</u> $T^{(k)}$. It is described algorithmically as follows:

```
SIMPLE PRICING WITH THRESHOLD:
1:  REPEAT FOR $\ell$ FROM 1 TO $t$:
    1.1:  REPEAT FOR $j$ nonbasic in period $p_\ell^{(k)}$:
              IF $d_j \geq T^{(k)}$: Select $j$ to enter basis; RETURN
    1.2:  CHOOSE $q_\ell$ such that $d_{q_\ell} > d_j$ for all $j$ in period $p_\ell^{(k)}$
          IF $d_{q_\ell} > \tau^{(k)}$: Select $q_\ell$ to enter basis; RETURN
2:  CHOOSE $q$ such that $d_q > d_{q_\ell}$ for $\ell = 1,\ldots,t$
    IF $d_q \geq \varepsilon$: Select $q$ to enter basis; RETURN
3:  Declare basis optimal; RETURN
```

$T^{(k)}$ must be determined along with $\tau^{(k)}$ at each iteration. Experiments of Section 6 use the formula

$$T^{(k)} = \Gamma(\tau^{(k)}/\gamma)$$

where γ is the parameter for the $\tau^{(k)}$ update algorithm above. Since $\tau^{(k)}$ is basically γ times a running average of previous reduced costs, $T^{(k)}$ is a multiple Γ of the same running average. A Γ of 1.1 was used for most tests in Section 6.

<u>Continuous pricing by period</u>

It is fastest and simplest to price the variables of each period in some fixed order. As a result, however, simple pricing with theshold tends to favor the earlier variables in each period: a later variable is priced only if $d_j < T^{(k)}$ for all of the earlier ones.

To remedy this situation, the pricing algorithm may be modified so that it always continues where it left off at the preceding iteration. More precisely, suppose that variable $q^{(k-1)}$ of period $p^{(k-1)}$ was chosen to enter the basis at period k-1. Then the revised algorithm, called <u>continuous</u> pricing by period, proceeds as follows:

```
CONTINUOUS PRICING:
0:  IF $q^{(k-1)}$ is not the last variable in period $p^{(k-1)}$:
    0.1:  REPEAT FOR $j$ nonbasic in period $p^{(k-1)}$ FROM $q^{(k-1)}+1$:
              IF $d_j \geq T^{(k)}$: Select $j$ to enter basis; RETURN
    0.2: CHOOSE $q_0$ such that $d_{q_0} \geq d_j$ for all j priced
         in step 0.1
         IF $d_{q_0} \geq \tau^{(k)}$: Select $q_0$ to enter basis; RETURN
1: REPEAT FOR $\ell$ FROM 1 TO t:
    1.1: REPEAT FOR j nonbasic in period $p_\ell^{(k)}$ (TO $q^{(k-1)}$
         IF $p_\ell^{(k)} = p^{(k-1)}$):
            IF $d_j \geq T^{(k)}$: Select j to enter basis; RETURN
    1.2: CHOOSE $q_\ell$ such that $d_{q_\ell} \geq d_j$ for all j in period $p_\ell^{(k)}$
         IF $d_{q_\ell} \geq \tau^{(k)}$: Select $q_\ell$ to enter basis; RETURN
2: CHOOSE q such that $d_q \geq d_{q\ell}$ for $\ell = 0, \ldots, t$
   IF $d_q \geq \epsilon$: Select q to enter basis; RETURN
3: Declare basis optimal; RETURN
```

Repeated pricing by period

A special case of continuous pricing always chooses the first-priced period of the current iteration, $p_1^{(k)}$, to be the last-priced period of the preceding iteration, $p^{(k-1)}$. It is easily seen that the effect of such a strategy is to price all of period $p^{(k-1)}$ before any of the other periods, in a cyclic fashion. As a consequence the incoming variable is likely to again lie in $p^{(k-1)}$. Indeed, the incoming variable will be chosen repeatedly from $p^{(k-1)}$ until $d_j < \tau^{(k)}$ for every j in that period.

This idea of repeated pricing by period can be implemented as a separate (though similar) algorithm, as follows:

REPEATED PRICING

```
0:
    0.1: REPEAT FOR $j$ nonbasic in period $p^{(k-1)}$,
         FROM $q^{(k-1)}+1$ TO end, and FROM start TO $q^{(k-1)}$:
           IF $d_j \geq T^{(k)}$: Select $j$ to enter basis; RETURN
    0.2: CHOOSE $q_0$ such that $d_{q_0} \geq d_j$ for all $j$ in period $p^{(k-1)}$
         IF $d_{q_0} \geq \tau^{(k)}$: Select $q_0$ to enter basis; RETURN
1:  REPEAT FOR $\ell$ FROM 1 TO t (UNLESS $p_\ell^{(k)} = p^{(k-1)}$):
      [same step 1 as simple pricing with threshold]
2:  CHOOSE $q$ such that $d_q \geq d_{q_\ell}$ for $\ell = 0, \ldots, t$
    IF $d_q \geq \epsilon$: Select $q$ to enter basis; RETURN
3:  Declare basis optimal; RETURN
```

Pricing cyclically

It remains to specify how the periods will be ordered for any of these four algorithms. Computational experience has shown that choice of an ordering is often critical; sometimes running times vary much more between different orderings than between different algorithms.

An obvious "neutral" ordering is a cyclic one that starts wherever pricing left off at the previous iteration. Writing $p^{(k-1)}$ for the last-priced period at iteration k-1, a "forward" cyclic order at iteration k prices

$$p^{(k-1)}+1,\ p^{(k-1)}+2,\ \ldots\ ,\ t,\ 1,\ 2,\ \ldots\ ,\ p^{(k-1)}-1,\ p^{(k-1)}$$

or, more formally,

$$p_\ell^{(k)} = \begin{cases} p^{(k-1)} + \ell & \text{if } p^{(k-1)} + \ell \le t \\ p^{(k-1)} + \ell - t & \text{if } p^{(k-1)} + \ell > t \end{cases}$$

Similarly a "backward" cyclic order is $p^{(k-1)}-1, \ldots, 1, t, \ldots, p^{(k-1)}$.

Cyclic orderings are suitable to any of the above algorithms. Simple or continuous pricing with a cyclic order should discourage suboptimization, since the candidate set is rotated among all the periods. Repeated pricing, on the other hand, tends to favor suboptimization since it prices the same period repeatedly. With a cyclic ordering, however, repeated pricing always moves on to period $\ell+1$ when it has finished with period ℓ, so at least the suboptimization proceeds cyclically through all the periods.

Many partial-pricing methods for general LPs use a kind of cyclic ordering: if $j^{(k-1)}$ was the last column priced at iteration k-1, then pricing at iteration k begins with column $j^{(k-1)}+1$ (or with column 1 if $j^{(k-1)}$ is the last column of the LP), and continues cyclically through all the nonbasic columns until some stopping rule is invoked. Cyclic continuous pricing by period is quite similar, although its stopping rule does take the periods into account.

If the price vector is only partially computed as suggested in Section 3, then forward cyclic pricing is preferable. At the start of pricing, the vector π is computed for periods $t, \ldots, p^{(k-1)}+1$ only; in most cases no more of π will be needed. The remainder of π is computed only at iterations in which period 1 is priced.

Since extra computation is required at period 1, it may pay to apply a weaker tolerance $\bar{\tau}^{(k)} \le \tau^{(k)}$ after periods $p^{(k-1)}+1, \ldots, t$

have been priced: if $d_j \geq \bar{\tau}^{(k)}$ for any variable in these periods, the best candidate so far is selected and pricing of period 1 is put off to the following iteration. The tests in Section 6 use the following heuristic formula:

$$\bar{\tau}^{(k)} = \tau^{(k)}/(p^{(k-1)}+1)$$

The size of the reduction from $\tau^{(k)}$ to $\bar{\tau}^{(k)}$ is thus tied inversely to $p^{(k-1)}$, the number of additional periods for which π must be computed to price period 1.

<u>Pricing earliest or latest</u>

The simplest possible ordering starts with the first period and runs forward:

$$p_\ell^{(k)} = \ell \quad , \qquad \ell = 1,\ldots, t$$

A similar approach starts at the end and works backwards:

$$p_\ell^{(k)} = t + 1 - \ell \;,\; \ell = 1,\ldots, t$$

With simple pricing, these orderings produce an acceptable candidate from the earliest or latest possible period. They seem less suited to the other algorithms, with which they have a more complicated behavior.

Earliest and latest pricing should both tend to suboptimize heavily and consistently. They approximate a simple, intuitive strategy: first optimize the beginning (or end), then work forward (or backward). Success of such a strategy requires that the suboptima not be too far from a true optimum.

Pricing for a balanced basis

It was observed in Section 1 that staircase bases tend to be "well-balanced," in the sense that $n_t - m_t$--the excess of period-t columns over period-t rows--cannot be very far from zero. Partial pricing can encourage a balanced basis by trying to bring columns into the basis in column-deficient periods; if some of the outgoing columns happen to be in column-excessive periods, the overall balance of the basis will improve.

These ideas suggest ordering the periods so that the most column-deficient come first, and the most column-excessive are last. More formally, such an ordering satisfies the following relations (with $m_t^{(k)}$, $n_t^{(k)}$ being the row and column counts at iteration k):

$$p = p_1^{(k)} \Leftrightarrow n_p^{(k)} - m_p^{(k)} \le n_t^{(k)} - m_t^{(k)} \quad , \quad t = 1, \ldots, T$$

$$p = p_\ell^{(k)} \Leftrightarrow n_p^{(k)} - m_p^{(k)} \le n_t^{(k)} - m_t^{(k)} \quad , \quad t = 1, \ldots, T;$$
$$t \ne p_1^{(k)}, \ldots, p_{\ell-1}^{(k)}$$

If several periods are tied with the same value of $n_t^{(k)} - m_t^{(k)}$, they can be ordered among themselves in any way; it may be wise to use a cyclic ordering for this purpose, so that no period is unduly favored.

This ordering may be used with any of the pricing algorithms. In all cases the effect should be to keep $|n_t - m_t|$ small in all periods. As a consequence $\sum_1^t (n_i - m_i)$ should also tend to be small; and an increase in square sub-staircases (where $\sum_1^t (n_t - m_t) = 0$) should be expected. If pricing for balance does encourage square sub-staircases, it will of course also favor suboptimization.

Another possible advantage of a balanced basis arises when the staircase bump-and-spike techniques of [1,2] are employed for Gaussian elimination. A well-balanced basis has relatively few "interperiod spikes", and so its L and U factors (especially the latter) would be sparser. Routines for solving linear systems should operate faster as a result.

Pricing to avoid suboptimization

As explained previously, square sub-staircases (upper or lower) give rise to pricing barriers; when any column is brought into the basis it is effectively suboptimizing periods between the nearest preceding lower barrier and the nearest succeeding upper barrier. If the locations of barriers can be identified, partial pricing may seek to discourage suboptimization by favoring periods in the larger suboptimization intervals. Details are the same as for balanced pricing above, except that the number $n_t - m_t$ is replaced by the number of periods between the preceding lower and succeeding upper barriers nearest to period t.

In practice a basis may have numerous staircase forms, and it seems a hopeless task to keep track of the barriers in every one. It is much more reasonable to try to determine the upper and lower barriers in one staircase form, as defined in Section 4; this would require at most keeping track of the numbers of linking and non-linking columns in each period of the basis, as the numbers of rows are fixed. Storing this information and checking for barriers can be organized fairly efficiently. Unfortunately, it pertains only to the staircase form that the basis inherits from the LP; barriers in the corresponding reduced-staircase form (Section 1) may go undetected.

If the column-upper-staircase form of the basis is monitored, upper barriers will occur between the linking and non-linking columns of the same (lower-staircase) period. Thus it can be advantageous to consider the basis as having $2t - 1$ "half-periods": half-period $2\ell - 1$ comprises the non-linking columns of period ℓ, and half-period 2ℓ the linking columns. Half-periods are then ordered as above, and any of the algorithms may be modified to price a half-period instead of a period at a time. (The tests in Section 6--which monitored lower and column-upper barriers only--used a scheme of this sort.)

Barrier orderings are suitable to either simple or continuous pricing by period. There is not much point in using these orderings with repeated pricing, which tends to encourage rather than discourage sub-optimization.

6. COMPUTATIONAL EXPERIENCE

This section reports on initial computational experiments with some of the preceding suggestions for partial pricing by period. Results show, not surprisingly, that partial pricing is considerably superior to full pricing when reduced cost is the only pricing criterion. Tests also confirm that partial computation of the price vector affords noticeable savings.

Most importantly, in several cases best results are achieved by specialized staircase pricing methods that differ substantially from the common general methods. Partial pricing by period thus appears to offer savings not otherwise available. Such a conclusion is further borne out by comparisons with the performance of a standard commerical LP code.

Experimental setup

For the test runs an existing LP code, MINOS [15,17], was modified to recognize staircase structure and to apply staircase techniques. Since MINOS employs a bump-and-spike approach for sparse Gaussian elimination, the staircase bump-and-spike technique [2] was added as an option in the test version. Various optional pricing algorithms were also added, but in such a way that all use the same main loop for actually computing reduced costs. Timings for test runs with different options can thus be meaningfully compared.

The test code is programmed in FORTRAN. Further details of the code and the experimental setup are in Appendix B.

Owing to a limited (though large) computing budget, testing was confined to the following seven methods of partial pricing by period:

Ordering of periods	Algorithm
cyclic	simple
cyclic	continuous
cyclic	repeated
earliest	simple
latest	simple
balanced	simple with threshold
barrier	simple with threshold

Different values of tolerance parameter γ were first tested with one of the cyclic orderings: the γ that gave best results was then used for the other methods. The threshold parameter Γ was generally fixed at 1.1; a few tests at lower values gave no better results.

Six medium-scale linear programs of dissimilar proportions were employed in the experiments. Their overall dimensions are as follows:

	PERIODS	ROWS	COLUMNS	NONZEROES
SCARGR25	25	472	500	2208
SCRS8	16	491	1169	4106
GROW15	15	301	645	5666
SCFXM3	12	991	1371	8204
SCTAP2	10	1091	1880	8645
SCSD8	39	398	2750	11349

All experiments measured the total time and iterations to find an optimal solution from a feasible starting basis; the test code could just as well have started from an infeasible basis, but a feasible start was more economical and made the results easier to interpret. The starting bases were themselves produced from an all-slack start, using full pricing so that no method of partial pricing would be favored. Additional information about the test LPs is collected in Appendix A.

In interpreting the results it should be kept in mind that total iterations are a somewhat stochastic quantity that may vary by as much as 10% when even small changes are made. Proportionally small differences in iteration totals should thus not be taken too seriously. Many tables give both iterations and seconds per 100 iterations, so that it can be seen clearly how much of any improvement is due to iteration count and how much to iteration speed.

Run times are also imprecise, but much less so. Times presented here have been rounded to avoid false precision, but all ratios and percentages have been calculated from the original readings.

Overall results

As a point of reference, the test problems were first run with full greatest-reduced-cost pricing. All were tried both with and without a simple geometric-mean scaling (described in Appendix B): SCRS8, GROW15 and SCTAP2 showed notably fewer iterations scaled, while SCAGR25 and SCSD8 required fewer iterations unscaled. SCFXM3 needed about the same numbers of iterations scaled and unscaled, but the scaled version was preferred because it had a much smaller range of coefficient magnitudes.

Results of full pricing with the preferable scalings were as follows:

		FULL PRICING		
	SCALED?	ITERATIONS	CPU SEC/100 ITER	TOTAL CPU SEC
SCAGR25	NO	296	4.8	14.3
SCRS8	YES	342	6.3	21.6
GROW15	YES	572	5.8	32.9
SCFXM3	YES	478	10.5	50.3
SCTAP2	YES	540	10.6	57.2
SCSD8	NO	900	10.0	90.0

In informal tests scaling seemed to affect partial pricing in the same way as full pricing, and so the above choices of scaling or no scaling were kept throughout the tests.

Experiments with the parameter γ yielded the following settings:

	γ
SCAGR25	.5
SCRS8	.2
GROW15	.2
SCFXM3	.2
SCTAP2	.05
SCSD8	.5

As expected, a higher γ tended to reduce the number of iterations but to increase the cost per iteration. SCSD8 was an extreme case (with continuous cyclic pricing):

γ	iterations	cpu sec/100 iter	total cpu sec
1.0	906	7.0	63.1
0.5	998	3.9	38.6
0.2	1338	3.6	48.4

In other cases the choice of γ was not so critical. SCTAP2 seemed to require about the same number of iterations for any γ in the range 0.5-0.01, and so its best setting was smaller than the average.

The best partial-pricing test runs--in terms of total CPU seconds--were as follows:

	BEST RUNS			CPU SEC/	TOTAL	% OF FULL
	ORDER	METHOD	ITERATIONS	100 ITER	CPU SEC	PRICING
SCAGR25	balanced	simple w/thr	333	3.9	13.0	91%
	cyclic	continuous	348	3.9	13.5	94%
SCRS8	latest	simple	272	4.5	12.2	56%
GROW15	cyclic	repeated	508	3.8	19.4	59%
SCFXM3	cyclic	repeated	514	7.0	35.9	71%
SCTAP2	cyclic	continuous	614	6.4	39.1	68%
	cyclic	simple	642	6.1	39.4	69%
SCSD8	cyclic	repeated	996	3.8	38.1	42%
	cyclic	continuous	998	3.9	38.6	43%

The diversity of best methods is striking: of the seven methods tested, five showed up as best or near-best on at least one problem. Significantly, the one method most similar to non-staircase methods--cyclic continuous pricing--was never a unique best, and was not among the best at all in half the cases. Thus a strong case is made for staircase pricing by period.

Four of the six problems showed the expected behavior of partial pricing versus full pricing: an increase in number of iterations, but a compensating decrease in time per iteration. SCRS8 and GROW15, however, show a decrease over full pricing in both iterations and time per iteration, and their total running times are reduced dramatically. The only greater reduction is for SCSD8, whose timings for full pricing are inflated by its huge number of columns.

The reduction of iterations for SCRS8 and GROW15 is accomplished in both instances by pricing methods that tend to suboptimize. It thus appears that in some cases suboptimization is a highly successful strategy: it simultaneously shortens the iteration path and makes each iteration cheaper.

Overall, partial pricing was decisively superior to full pricing for all but the smallest LP, SCAGR25. Not too much should be made of this comparison, however, since full pricing by greatest reduced cost is seldom used in practice. It is more meaningful to compare these results with non-staircase partial pricing, as is done at the end of this section. A comparison with steepest-edge pricing (which is always full) would also be revealing, but unfotunately MINOS is not set up for the required calculations. Nevertheless, it does seem unlikely that a steepest-edge criterion could reduce the number of iterations for SCRS8, GROW15 or SCSD8 sufficiently to overcome the efficiencies of partial pricing.

Best results aside, different methods varied considerably in how close they came to best over the range of problems. Each method is examined individually below. At the end of this section the best times are again considered, in comparisons with non-staircase methods.

Cyclic pricing by period

Simple cyclic pricing by period generally gave acceptable, if not impressive, results:

SIMPLE CYCLIC PRICING

	ITERATIONS	CPU SEC/ 100 ITER	TOTAL CPU SEC	% OVER BEST RUN
SCAGR25	415	3.6	15.0	15%
SCRS8	429	4.0	17.2	42%
GROW15	588	4.8	28.0	44%
SCFXM3	510	7.6	38.7	8%
SCTAP2	642	6.1	39.4	1%
SCSD8	969	4.3	41.3	8%

Here the typical tradeoff with full pricing--more iterations, less time per iteration--is found in every case. Thus worst results are with SCRS8 and GROW15, for which this tradeoff can be avoided by other methods as discussed above.

Continuous cyclic pricing improved upon the simple version in every instance:

CONTINUOUS CYCLIC PRICING

	ITERATIONS	CPS SEC/ 100 ITER	TOTAL CPU SEC	% OVER BEST RUN
SCAGR25	348	3.9	13.5	4%
SCRS8	355	4.0	14.3	18%
GROW15	600	4.5	27.2	40%
SCFXM3	526	7.3	38.3	7%
SCTAP2	614	6.4	39.1	0%
SCSD8	998	3.9	38.6	1%

Thus the more sophistciated stopping rule of continuous pricing does pay off. Continuous cyclic pricing was most uniformly reliable of the methods tested, and should be preferred when no time is available for testing other methods.

By contrast, the performance of repeated cyclic pricing was somewhat mixed:

	REPEATED CYCLIC PRICING			
	ITERATIONS	CPU SEC/ 100 ITER	TOTAL CPU SEC	% OVER BEST RUN
SCAGR25	396	3.7	14.6	12%
SCRS8	375	4.0	15.2	25%
GROW15	508	3.8	19.4	0%
SCFXM3	514	7.0	35.9	0%
SCTAP2	1115	6.1	68.4	75%
SCSD8	996	3.8	38.1	0%

Repeated pricing is best in half the cases, but significantly worse in the other half. In the worst case, SCTAP2, it would be a disaster. Yet time per iteration is consistently no greater than for continuous pricing; the determining factor is number of iterations.

The behavior of repeated pricing confirms what one would expect of a method that tends to suboptimize. When suboptimization works well, it finds a short iteration path at low cost; when it works poorly it tends to get stuck at suboptimal solutions and the iteration path may be unduly long.

Earliest and latest pricing by period

Signs of suboptimization are especially clear in the results of simple earliest and simple latest pricing:

SIMPLE EARLIEST PRICING

	ITERATIONS	CPU SEC/ 100 ITER	TOTAL CPU SEC	% OVER BEST RUN
SCAGR25	467	4.2	19.5	50%
SCRS8	879	4.7	41.6	242%
GROW15	605	4.4	26.7	38%
SCFXM3	625	8.9	55.3	54%
SCTAP2	>1479	(8.0)	>120.0	>200%
SCSD8	>2000	(4.0)	> 79.7	>100%

SIMPLE LATEST PRICING

	ITERATIONS	CPU SEC/ 100 ITER	TOTAL CPU SEC	% OVER BEST RUN
SCAGR25	391	3.8	14.9	14%
SCRS8	272	4.5	12.2	0%
GROW15	1387	4.6	63.7	228%
SCFXM3	587	7.7	45.0	25%
SCTAP2	1200	7.1	84.9	117%
SCSD8	>2000	(5.7)	>112.9	>200%

The pattern is similar to that for repeated cyclic pricing, but more pronounced: huge variations in performance from problem to problem, with number of iterations the most important factor. Three runs were so hopelessly long that they were stopped prior to optimality; lower bounds on their results are indicated by notations such as ">2000".

Curiously, earliest pricing is much better than latest on half the problems, while latest is much better than earliest on the other half. There seems no obvious explanation for this dichotomy, except to say that it is due to the different natures of the problems.

Results are probably also strongly influenced by the choice of starting basis. In the case of SCTAP2, for instance, early tests showed that simple latest pricing is much superior to simple cyclic pricing when the initial basis is all-slack. Indeed, latest pricing required only 1177 iterations from an all-slack start, compared to 1200 from the feasible start above!

Clearly pricing methods that suboptimize can give spectacular results. But they are not uniformly effective, and only preliminary testing can determine whether they are appropriate to a particular LP.

Balanced pricing by period

The results of simple balanced pricing with threshold were for the most part uniformly mediocre:

SIMPLE BALANCED PRICING (WITH THRESHOLD)

	ITERATIONS	CPU SEC/ 100 ITER	TOTAL CPU SEC	% OVER BEST RUN
SCAGR25	333	3.9	13.0	0%
SCRS8	323	4.7	15.1	25%
GROW15	634	5.0	31.5	62%
SCFXM3	729	7.9	57.4	60%
SCTAP2	884	6.5	57.7	48%
SCSD8	1173	4.6	54.0	42%

Times per itteration are generally worse than for cyclic pricing; apparently there are not very many acceptable candidates in column-deficient periods, and more reduced costs must be computed. Iteration counts are also greater except for SCAGR25 and SCRS8. SCAGR25 is the only clear success.

Barrier pricing by period

Simple barrier pricing with threshold was worse than simple cyclic pricing in every instance:

SIMPLE BARRIER PRICING (WITH THRESHOLD)

	ITERATIONS	CPU SEC/ 100 ITER	TOTAL CPU SEC	% OVER BEST RUN
SCAGR25	442	4.1	17.9	38%
SCRS8	386	4.7	18.1	49%
GROW15	603	5.1	30.5	57%
SCFXM3	524	7.9	41.6	16%
SCTAP2	957	6.0	57.8	48%
SCSD8	1231	4.1	50.2	32%

Better times per iteration and fewer iterations were achieved by cyclic pricing in most cases.

Possibly barrier pricing employs too simple a criterion to be effective. As implemented in the test code, it looks only for square sub-staircases in the lower-staircase and column-upper-staircase forms inherited by the basis; it may overlook other pricing barriers. This hypothesis is supported somewhat by the following table, which lists average numbers of lower square sub-staircases in the reduced-staircase form of the basis, as reported by MINOS's basis-factorization routine:

MEAN LOWER SQUARE SUB-STAIRCASES

	FULL	CYCLIC (CONTINUOUS)	CYCLIC (REPEATED)	EARLIEST	LATEST	BALANCED	BARRIER
SCAGR25	1.7	2.8*	3.7	3.6	3.3	3.1*	2.0
SCRS8	4.0	2.1	1.2	1.2	5.2*	2.9	2.0
GROW15	9.7	11.6	12.0*	10.9	9.8	9.9	11.1
SCFXM3	2.4	2.8	2.5*	3.0	3.8	2.7	2.4
SCTAP2	4.9	3.9*	2.8	1.5	3.3	2.2	1.8
SCSD8	2.2	2.7*	8.2*	1.5	16.4	8.7	3.5

*Best runs

The figures vary considerably from one pricing method to the next, yet no strong pattern emerges. One may conclude that pricing strongly influences the barrier structure of the basis, but in a complex way.

Partial computation of the price vector

All of the runs reported to this point used a staircase "bump-and-spike" pivot order in Gaussian elimination, and hence computed only part of the price vector as described in Section 3. To determine the value of this arrangement, six of the best runs were duplicated twice with full computation of π: once using the same staircase pivot order, and once with the standard pivot order.

Although theoretically the pivot order and computation of π should have no effect upon the iteration path, in practice small numerical differences can result in quite different paths and different numbers of iterations. Thus times for the full-π runs below were normalized to reflect the same number of iterations as the partial-π run.

Results of these tests were as follows:

			CPU SECONDS, π COMPUTED AS FOLLOWS:		
	ORDER	METHOD	PARTIAL: STAIR PIV	FULL: STAIR PIV	FULL: STANDARD PIV
SCARG25	balanced	simple w/thr	13.0	14.2 (- 8%)	13.9 (- 7%)
SCRS8	latest	simple	12.2	13.5 (-10%)	13.8 (-12%)
GROW15	cyclic	repeated	19.4	21.9 (-11%)	22.0 (-12%)
SCFXM3	cyclic	repeated	35.9	38.2 (- 6%)	38.6 (- 7%)
SCTAP2	cyclic	continuous	39.1	42.7 (- 8%)	42.0 (- 7%)
SCSD8	cyclic	continuous	38.6	44.1 (-12%)	45.0 (-16%)

Partial computation of π is seen to offer a modest but consistent saving of 6-12%. With full computation of π the staircase pivot order offers no advantage over the standard order except for SCSD8.

These findings confirm those of [2]. It is possible, as suggested in [2], that other staircase pivot-ordering techniques may handle some of these LPs better than the bump-and-spike technique; but the savings through partial computation of π should be realized equally well with any staircase ordering.

Comparison with a commercial code

To compare staircase pricing to a traditional approach, the six test problems were also solved with the WHIZARD LP code of the MPS III system [14]. WHIZARD is a commercially marketed assembly-language code that should be inherently faster than the FORTRAN test version of MINOS. However, a comparison of the iteration counts and timings should give some rough idea of the practicality of staircase pricing techniques.

WHIZARD employs a combination of partial and multiple pricing with a maximum-reduced-cost criterion. Optional scaling is available, and was used with the four LPs that were scaled for MINOS. Details of the WHIZARD runs appear at the end of Appendix B.

The best MINOS runs for each LP compare with the WHIZARD runs as follows:

	ITERATIONS		SEC/100 ITER		TOTAL SEC		MINOS VS
	MINOS	WHIZ	MINOS	WHIZ	MINOS	WHIZ	WHIZARD
SCAGR25	333	301	3.9	4.1	13.0	12.2	+ 7%
SCRS8	272	499	4.5	3.6	12.2	16.1	-24%
GROW15	508	788	3.8	3.3	19.4	26.1	-26%
SCFXM3	514	524	7.0	5.5	35.9	28.5	+26%
SCTAP2	614	658	6.4	4.5	39.1	29.4	+33%
SCSD8	996	1077	3.8	3.7	38.1	39.4	- 3%

Surprisingly, MINOS was faster in half the cases and significantly slower in only two. This favorable showing can be explained largely by the advantages of staircase pricing.

Staircase pricing under MINOS required substantially fewer iterations with SCRS8, GROW15 and SCSD8, offsetting any disadvantage in time per iteration. Hence MINOS was actually faster for these problems. Moreover, in the cases of GROW15 and SCSD8 the MINOS times per iteration were not much greater than the WHIZARD times; it appears that MINOS made up for its inherent slowness by computing only part of π and by pricing far fewer variables per iteration. (It is hard to say anything more precise, however, since WHIZARD does not report the number of variables priced.)

SCAGR25 reversed the situation: it traded more iterations under MINOS for slightly <u>less</u> time than WHIZARD per iteration. On the whole MINOS came out only slightly behind.

For SCFXM3 and SCTAP2 the numbers of iteration were comparable and WHIZARD had the expected edge in time per iteration. As a result MINOS was about 30% slower overall, a respectable showing considering the inefficiencies of FORTRAN.

These results suggest that a truly fast implementation of staircase pricing--perhaps incorporating machine language in critical portions --would be advantageous in almost every case and highly advantageous in many cases. If the best methods of [2] were also implemented, some staircase LPs could well be solved in a half or even a quarter of the time currently taken by the fastest codes.

APPENDIX A: TEST PROBLEMS

The linear programs used for the computational experiments of Section 6 are described in greater detail below. The tabular summaries for each LP are largely self-explanatory, but a few general notes are appropriate:

All statistics except OBJ ELEMS refer only to the staircase constraint matrix, excluding the objective row and right-hand side. In each case the constraint matrix, A, has been put in reduced standard form; DIAGONAL BLOCKS refers to the staircase blocks $A_{\ell\ell}$, and OFF-DIAGONAL BLOCKS to the blocks $A_{\ell+1,\ell}$.

Variables (columns) are implicitly constrained only to be non-negative, unless there is an indication to the contrary. BOUNDED implies implicit lower and upper bounds.

MAX ELEM and MIN ELEM are the largest and smallest magnitudes of elements in A; LARGEST COL RATIO is the greatest ratio of magnitudes of any two elements in the same column of A. Where values are given BEFORE SCALING and AFTER SCALING, all tests were conducted with A scaled as described in Appendix B. Otherwise NO SCALING is indicated.

SCAGR25

Test problem received from James K. Ho, Brookhaven National Laboratory, Upton, N.Y.; source not documented.

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1	18	20	45	13%	8	7	17	30%	19
2-24	19	20	46	12%	8	7	17	30%	19
25	16	20	43	13%					19
			1146	12%			408	30%	475

GRAND TOTALS		
ROWS	471	(300 EQUALITIES, 171 INEQUALITIES)
COLS	500	
ELEMS	1554	
DENS	0.7%	

COEFFICIENTS	NO SCALING
MAX ELEM	9.3
MIN ELEM	2.0×10^{-1}
LARGEST COL RATIO	1.9×10^{1}

SCRS8

Derived from a model of the United States' options for a transition from oil and gas to synthetic fuels; documented in [9,13].

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1	28	37	65	6%	25	22	29	5%	18
2	28	38	69	6%	25	22	29	5%	19
3-5	31	76	181	8%	25	22	29	5%	55
6-8	32	79	192	8%	25	22	29	5%	58
9	31	79	189	8%	25	22	29	5%	58
10-12	31	80	190	8%	25	22	29	5%	59
13-15	30	80	186	8%	25	22	29	5%	59
16	31	70	177	8%					59
			2747	8%			435	5%	847

GRAND TOTALS

ROWS	490	(384 EQUALITIES, 106 INEQUALITIES)
COLS	1169	
ELEMS	3182	
DENS	0.6%	

COEFFICIENTS	BEFORE SCALING	AFTER SCALING
MAX ELEM	1.9×10^{2}	4.0
MIN ELEM	1.0×10^{-3}	2.5×10^{-1}
LARGEST COL RATIO	4.5×10^{3}	1.6×10^{1}

GROW15

A simple dynamic input-output LP constructed for test purposes. It is based on the following model: define sets,

IND	set of goods
OBJ	set of export goods: a subset of IND

and parameters,

T	number of periods
a_{ij}	units of good i needed to produce 1 unit of good j; $i, j \in$ IND
m_i	maximum production of good i in a period; $i \in$ IND
σ	proportion of m_i that may be stored; σm_i is maximum stock of good i at beginning of a period
p_{it}	expected income per unit of good i in period t; $i \in$ OBJ, $t = 1,\ldots, T$

Then the variables are

x_{it}	production of good i in period t; $i \in$ IND, $t = 1,\ldots, T$
s_{it}	stock of good i at beginning of period t; $i \in$ IND, $t = 1,\ldots, T+1$

and the LP is

maximize $\sum_{t=1}^{T} \sum_{i \in OBJ} p_{it} \; x_{it}$

subject to $s_{i,t+1} = s_{it} + x_{it} + \sum_{j \in IND} a_{ij} x_{jt}$, $i \in IND, \; t = 1,\ldots,T$

$0 \le x_{it} \le m_i$ $i \in IND, \; t = 1,\ldots,T$

$0 \le s_{it} \le \sigma m_i$ $i \in IND, \; t = 1,\ldots,T+1$

For GROW15, T = 15. The values of a_{ij} and m_i were taken from a 20-sector input-output analysis of the U. S. economy in [3]. A set OBJ of size 3 was picked arbitrarily, as were the values p_{it}; σ was set at 0.3.

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1-14	20	43	356	41%	20	20	20	5%	3
15	20	43	356	41%					3
			5340	41%			280	5%	45

GRAND TOTALS

ROWS	300	(ALL EQUALITIES)
COLS	645	(510 BOUNDED)
ELEMS	5620	
DENS	2.9%	

COEFFICIENTS	BEFORE SCALING	AFTER SCALING
MAX ELEM	1.0	1.2×10^{2}
MIN ELEM	6.0×10^{-6}	8.1×10^{-3}
LARGEST COL RATIO	1.3×10^{5}	1.5×10^{4}

SCFXM3

Test problem received from James K. Ho, Brookhaven National Laboratory, Upton, N.Y.; source not documented.

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1	92	114	679	6%	9	57	61	12%	13
2	82	99	434	5%	9	35	35	11%	4
3	66	126	300	4%	5	33	33	20%	1
4	90	118	1047	10%	5	5	5	20%	5
5	92	114	679	6%	9	57	61	12%	13
6	82	99	434	5%	9	35	35	11%	4
7	66	126	300	4%	5	33	33	20%	1
8	90	118	1047	10%	5	5	5	20%	5
9	92	114	679	6%	9	57	61	12%	13
10	82	99	434	5%	9	35	35	11%	4
11	66	126	300	4%	5	33	33	20%	1
12	90	118	1047	10%					5
			7380	7%			397	13%	69

GRAND TOTALS		
ROWS	990	(561 EQUALITIES, 429 INEQUALITIES)
COLS	1371	
ELEMS	7777	
DENS	0.6%	

COEFFICIENTS	BEFORE SCALING	AFTER SCALING
MAX ELEM	1.3×10^{2}	1.1×10^{1}
MIN ELEM	5.0×10^{-4}	8.7×10^{-2}
LARGEST COL RATIO	1.3×10^{5}	1.3×10^{2}

SCTAP2

A dynamic traffic assignment problem, documented in [10]. The LP has 11 objective rows; the one named OBJZZZZZ was used in all tests, and the other ten were deleted. Statistics below omit the ten deleted objective rows.

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1-9	109	188	423	2%	62	138	276	3%	141
10	109	188	423	2%					141
			4230	2%			2484	3%	1410

GRAND TOTALS

ROWS	1090	(470 EQUALITIES, 620 INEQUALITIES)
COLS	1880	
ELEMS	6714	
DENS	0.3%	

COEFFICIENTS	BEFORE SCALING	AFTER SCALING
MAX ELEM	8.0×10^{1}	2.5
MIN ELEM	1.0	4.0×10^{-1}
LARGEST COL RATIO	8.0×10^{1}	6.4

SCSD8

A multi-stage structural design problem, documented in [8]. This is the only staircase test problem in which the states do not represent periods of time.

	DIAGONAL BLOCKS				OFF-DIAGONAL BLOCKS				OBJ
PERIOD	ROWS	COLS	ELEMS	DENS	ROWS	COLS	ELEMS	DENS	ELEMS
1-38	10	70	130	19%	10	50	90	18%	70
39	17	90	224	15%					90
			5164	18%			3420	18%	2750

GRAND TOTALS

ROWS	397	(ALL EQUALITIES)
COLS	2750	
ELEMS	8584	
DENS	0.8%	

COEFFICIENTS	NO SCALING
MAX ELEM	1.0
MIN ELEM	2.4×10^{-1}
LARGEST COL RATIO	4.0

APPENDIX B: DETAILS OF COMPUTATIONAL TESTS

Computing environment

All computational experiments were performed on the Triplex system [19] at the Stanford Linear Accelerator Center, Stanford University. The Triplex comprises three computers linked together: one IBM 360/91 and two IBM 370/168s. Runs were submitted as batch jobs in a virtual-machine environment, under the control of IBM systems OS/VS2, OS/MVT and ASP.

Test runs employed a specially-modified set of linear-programming routines from the MINOS system [15,17]. MINOS is written in standard FORTRAN. For timed runs, MINOS was compiled with the IBM FORTRAN IV (H extended, enhanced) compiler, version 1.1.0, at optimization level 3 [11].

Timings

All running-time statistics are based on "CPU second" totals for individual job steps as reported by the operating system. To promote consistency all timed jobs were run on the Triplex computer designated "system A," and jobs whose timings would be compared were run at about the same time. Informal experiments showed roughly a 1% variation in timings due to varying system loads.

MINOS linear-programming environment

MINOS was set up for test runs according to the defaults indicated in [15], with the exception of the items listed below.

Scaling. Problems noted as "scaled" in Appendix A were subjected to the following geometric-mean scaling (where A denotes the matrix of constraint coefficients, not including the objective or right-hand side):

1: Compute $\rho_0 = \max|A_{i_1 j}/A_{i_2 j}|$, $A_{i_2 j} \neq 0$.

2: Divide each row i of A, and its corresponding right-hand side value, by $[(\min_j |A_{ij}|)(\max_j |A_{ij}|)]^{1/2}$, taking the minimum over all $A_{ij} \neq 0$.

3: Divide each column j of A, and its corresponding coefficient in the objective, by $[(\min_i |A_{ij}|)(\max_i |A_{ij}|)]^{1/2}$, taking the minimum over all $A_{ij} \neq 0$.

4: Compute $\rho = \max|A_{i_1 j}/A_{i_2 j}|$, $A_{i_2 j} \neq 0$.

This procedure was repeated as many times as possible until, at step 4, ρ was at least 90% of ρ_0. (In other words, scaling continued as long as it reduced ρ, the greatest ratio of two magnitudes in the same column, by more than 10%.)

Starting basis. A feasible starting basis was determined for each LP as follows. MINOS was slightly modified so that it would stop and save the first feasible basis obtained; each LP except GROW15 was run on this modified version, from an all-slack start (crash option 0) with full pricing. The saved feasible basis was used as a starting basis in all subsequent test runs. For GROW15 an all-slack basis is feasible, and so all test runs of GROW15 employed an all-slack (crash option 0) start.

Termination. Virtually all test runs terminated at an optimal solution. However, three runs--as indicated in Section 6--were terminated short of optimality because they required too much time or too many iterations.

Basis factorization. The staircase bump-and-spike factorization of [2] was employed in all test runs except as indicated otherwise in Section 6.

Refactorization frequency. The "INVERT FREQ" for MINOS was set to 50; hence MINOS refactorized the basis (by performing a fresh Gaussian elimination) every 50 iterations.

Tolerances. The "LU ROW TOL" for MINOS was set to 10^{-4}. All other tolerances were left at their default values.

Modifications to MINOS

All MINOS runs described in this paper were made with a special test version of MINOS. This version was essentially the same as the special test version described in Appendix B of [2], except for modifications to subroutine PRICE to implement the algorithms of Section 5.

Modified subroutines that are particularly important to pricing are described briefly as follows:

BTRANL optionally computes only part of the price vector as outlined in Section 3. (There is no provision for updating the price vector.)

SETPI determines the price vector back to a specified period, calling BTRANL as necessary.

PRICE chooses a nonbasic variable to enter the basis, employing either full pricing or one of the partial-pricing methods described in Section 5. Reduced costs are computed from the price vector by method (a) of Section 2. SETPI is called one or more times to get needed parts of the price vector.

SPECS2 determines which pricing method and algorithm will be used in a particular run, and sets the parameters γ and Γ, according to instructions in the SPECS input file.

Other modifications are summarized in Appendix B of [2].

MPS III linear programming environment

For purposes of comparison all test problems were also run on the MPS III system [14], as explained in Section 6.

MPS III runs employed the WHIZARD linear-programming routines of version 8915 of MPS III. Starting bases were the same as for the MINOS runs, and termination was at an optimal solution in every case. CPU timings reported in Section 6 include both the compiler and executor steps.

The control program for a typical MPS III run was as follows:

```
         PROGRAM
         INITIALZ
         XPROC = XPROC + 6000
         XCLOCKSW=0
         XINVERT=1
         XFREQINV = 50
         XFREQLGO=1
         XFREQ1= 3000
         MVADR(XDQFREQ1,TIME)
         MOVE(XDATA,'SCRS8')
         MOVE(XPBNAME,'SCRS8')
         CONVERT('FILE','INPUT')
         SETUP('MIN','SCALE')
         MOVE(XOBJ,'COST')
         MOVE(XRHS,'RHS')
         INSERT('FILE','PUNCH1')
WHIZFREQ DC (250)
WHIZSCAL DC (4)
         WHIZARD('SCALE',WHIZSCAL)
TIME     EXIT
         PEND
```

Control programs for the other LPs were essentially the same. However, the 'SCALE' parameter was dropped from the SETUP and WHIZARD lines for SCAGR25 and SCSD8, since these two LPs were unscaled in all of the MINOS runs.

REFERENCES

[1] Fourer, Robert, "Sparse Gaussian Elimination of Staircase Linear Systems." Technical Report SOL 79-17, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1979).

[2] _____, "Solving Staircase Linear Programs by the Simplex Method, 1: Inversion." Technical Report SOL 79-18,, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1979).

[3] Glassey, C. Roger and Peter Benenson, "A Quadratic Programming Analysis of Energy in the United States Economy." Report ES-116, Electric Power Research Institute, Palo Alto, CA (1975).

[4] Goldfarb, D., "On the Bartels-Golub Decomposition for Linear Programming Bases." Mathematical Programming 13 (1977), 272-279.

[5] _____ and J. K. Reid, "A Practicable Steepest-Edge Simplex Algorithm." Mathematical Programming 12 (1977), 361-371.

[6] Harris, Paula M. J., "Pivot Selection Methods of the Devex LP Code." Mathematical Programming 5 (1973), 1-28.

[7] Herman, Richard J., "Dynamically Restricted Partial Pricing in the Simplex Method for Linear Programming." Report RC 7151, IBM Watson Research Center, Yorktown Heights, N.Y. (1978).

[8] Ho, James K., "Optimal Design of Multi-Stage Structures." Computers and Structures 5 (1975), 249-255.

[9] _____, "Nested Decomposition of a Dynamic Energy Model." Management Science 23 (1977), 1022-1026.

[10] _____, "A Successive Linear Optimization Approach to the Dynamic Traffic Assignment Problem." Report BNL-24713, Brookhaven National Laboratory, Upton, N.Y. (1978).

[11] IBM OS FORTRAN IV (H Extended) Compiler Programmer's Guide. No. SC28-6852, International Business Machines Corp. (1974).

[12] Kuhn, Harold W. and Richard E. Quandt, "An Experimental Study of the Simplex Method." Proceedings of Symposia in Applied Mathematics 15 (American Mathematical Society, 1963), 107-124.

[13] Manne, A. S., "U. S. Options for a Transition from Oil and Gas to Synthetic Fuels." Discussion Paper 26D, Public Policy Program, Kennedy School of Government, Harvard University (1975).

[14] MPS III Mathematical Programming System: User Manual. Ketron, Inc., Arlington, VA (1975).

[15] Murtagh, Bruce A. and Michael A. Saunders, "MINOS: A Large-Scale Nonlinear Programming System (For Problems with Linear Constraints)." Technical Report SOL 77-9, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1977).

[16] Orchard-Hays, William, Advanced Linear-Programming Computing Techniques (New York: McGraw-Hill Book Co., 1968).

[17] Saunders, Michael A., "MINOS System Manual." Technical Report SOL 77-31, Systems Optimization Laboratory, Dept. of Operations Research, Stanford University (1977).

[18] Tomlin, J. A., "On Pricing and Backward Transformation in Linear Programming." Mathematical Programming 6 (1974), 42-47.

[19] Vinson, Ilse, "Triplex User's Guide." User Note 99, SLAC Computing Services, Stanford Linear Accelerator Center (1968).

[20] Wolfe, Philip, "The Composite Simplex Algorithm." SIAM Review 7 (1965), 42-54.

[21] _____ and Leola Cutler, "Experiments in Linear Programming." Recent Advances in Mathematical Programming, R.L. Graves and Philip Wolfe, eds. (New York: McGraw-Hill Book Co., 1963), 177-200.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE — READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER (14) SOL-79-19	2. GOVT ACCESSION NO.
	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Solving Staircase Linear Programs By The Simplex Method, 2. Pricing.	5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report.
	6. PERFORMING ORG. REPORT NUMBER SOL 79-19
7. AUTHOR(s) (10) Robert/Fourer	8. CONTRACT OR GRANT NUMBER(s) (15) N00014-75-C-0267. [illegible]-SF[illegible]
9. PERFORMING ORGANIZATION NAME AND ADDRESS Operations Research Department - SOL Stanford University Stanford, CA 94305	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR-047-143
11. CONTROLLING OFFICE NAME AND ADDRESS Operations Research Program - ONR Department of the Navy 800 N. Quincy Street, Arlington, VA 22217	12. REPORT DATE (11) Nov[redacted]1979
	13. NUMBER OF PAGES 66 (12) 70
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

LARGE-SCALE LINEAR PROGRAMMING
STAIRCASE LINEAR PROGRAMS
SIMPLEX METHOD

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

SEE ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

[illegible]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

79-19, Robert Fourer
Solving Staircase Linear Programs by the Simplex Method, 2: Pricing

This and a companion paper share one goal: to solve staircase-structured linear programs faster through adaptation of the algorithms of the modern simplex method. Their means are quite different however: whereas the preceding paper concentrated on "inversion" algorithms that factorize the basis and solve linear systems, the present paper looks are "pricing" algorithms that select a variable to enter the basis at each iteration.

Pricing involves two sets of algorithms: computation algorithms that determine reduced costs of the nonbasic variables, and selection algorithms that choose among variables whose reduced costs are favorable. This paper develops staircase adaptations of both sorts of algorithms, and reports extensive (although preliminary) computational experience. Staircase computation algorithms appear to offer modest but consistent savings; staircase selection algorithms, properly chosen, may offer substantial savings in number of iterations, time per iteration, or sometimes both.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)